# Prologのコルーチン・インタプリタの検証について(同期の数理)

AUTHOR(S):

古川, 康一; 新田, 克己

CITATION:

古川, 康一 ...[et al]. Prologのコルーチン・インタプリタの検証について(同期の数理). 数理解析研究所講究録 1982, 470: 29-45

ISSUE DATE:

1982-10

URL:

http://hdl.handle.net/2433/103229

数理解析研究所講究録
470巻 1982年 29-45



# Prologのコルーチン・インタプリタの検証について

古川 康一　新田 克己
（電子技術総合研究所）

1. はじめに　Prologのインタプリタは，1971年にマルセーユ大学でcolmerauer等によって開発されて以来，現在までに国内外の各所で作られている。Prologインタプリタの特徴は，後戻り制御とユニフィケーションであり，それをいかにうまく実現するかが問われてきた。後戻り制御を実現するには，手続きの再帰呼出しだけでは不十分であり，それ以上の制御メカニズムが必要であることが知られていた。本稿では，コルーチン制御メカニズムを用いてPrologインタプリタの後戻り制御を実現した。インタプリタはAND/OR探索木の2種の節，すなわちAND節とOR節に対応した2種のプロセスから成り立っている。それらのプロセスはプログラムの解釈実行時に動的に生成・削除されて動く。本稿では，まず次節で，これら2種のプロセス（ANDプロセスと，ORプロセス）の記述を行い，その動きの概略を説明する。3節では，このインタプリタが正しく働くことを証明する。4節では，セミ・コルーチンによるインタプリタの記述との

等価性について述べる。最後に，コルーチン・インタプリタ上での種々の拡張について，簡単にふれる。

## 2. AND/OR プロセスの記述

AND/OR プロセスを記述するのに，本稿では，PAD（Problem Analysis Diagram）を用いる。PADは二村によって作られたフロー・チャートの一種であるが，記述が簡単で，労力が省ける（実際に，必要とする線分の数が，通常のフロー・チャートよりずっと少なくてすむ）のと同時に，プログラムの構造が，画上によく表現される。

PADによるANDプロセスの表現をFig.1に示す。ANDプロセスは，主プログラム，および各クローズの右辺の実行時に作成され，ゴール列の達成を行う。引数として受けとったゴール列の先頭から，順次ゴールを達成して行き，すべて達成したら，その旨を親に報告する。ここで，親は主プログラムを管理するコマンド・システムか，あるいはORプロセスの1つである。各ゴールの達成は，個々にORプロセスを作り，それらに委ねる。個々のORプロセスから成功の報告を受ければつぎのゴールに進んでよいが，失敗した場合には後戻り制御が必要になる。Fig.1の(L)以下がそのための処理

である。図のENTER(L)は，一度このANDプロセスが成功した後で他の所で失敗したときに，その失敗がこのプロセスにまで及んだときの再実行のための入口である。B-stackは，失敗した時にどこまで実行を遡上るかを記憶するためのスタックである。

ORプロセスのPAD表現をFig.2に示す。ORプロセスは個々のゴールに対して1つづつ作られ，そのゴールの達成のために働く。ORプロセスは，ゴールとマッチする候補手続きをプログラムの中から順に探してゆき，その手続き本体の達成を試みる。本体の実行は，そこでANDプロセスを1つ作り，それに委ねる。もしそれが成功なら自分も成功したことになり，その旨，親（ANDプロセス）に報告する。もし失敗したら，つぎの候補手続きを探して，ゴールの達成を再度試みる。すべての候補手続きが失敗したら，このORプロセス自身，失敗である。そのとき，親へその旨報告して，自分は消滅する。

## 3. コルーチン・インタプリタの検証

コルーチン・インタプリタの検証を，本稿では，制御部分とユニフィケーション部分に分けて行う。

### 3.1. 制御部分

はじめに，コルーチン・インタプリタがAND-OR木をすべての可能性について探索していくことを証明する。

**定理1.** Prolog プログラムのAND-OR木が有限であれば，対応するAND/ORプロセスは，それらが起動されるたびに解がある限り1つづつ解を生成し，最後は"失敗"で終る。

この証明は，AND-OR木の深さについての帰納法で行う。はじめに帰納法のベースとして，深さ0のANDプロセスおよび深さ1のORプロセスについて上の定理を証明する。

#### (1). 深さ0のANDプロセス

深さ0のANDプロセスは，空のゴール列を処理する。Fig.1のプログラムにおいて，curr_goalはnilとなるので，直ちに成功する。

もう一度起動されると，実行が $E_2$ から再開されるが，B-stackは空であるので，今回は失敗する。

ところが高さ0のand節はユニット・クローズなので，その解はそれ自身しかない。ゆえに，この場合は定理が成り立つ．

## (2). 深さ1のORプロセス

いま，高さ1のORプロセス O1が候補クローズ $C_1, \cdots, C_i$ を調べて，$C_1 \vee \cdots \vee C_i$ のすべての可能な解を生成したと仮定する。プログラムの制御は，Fig.2の m にある．そして，つぎの候補は $C_{i+1}$ である。

いま，ゴールが $C_{i+1}$ のクローズ・ヘッドとユニファイできたとする。すると，$C_{i+1}$ に対応するANDプロセス A1が作られ，起動される。それは高さが0のANDプロセスだから，前の証明より直ちに成功し，"while success" loopに入る。ここで解は親のANDプロセスに渡され，O1は $e_2$ で中断される。

つぎに再び起動されると，それはA1を再び起動する．今回は，直ちに失敗するので，制御は"delete curr_and"，"recover state"とすすみ，"for all" loopへ戻る．候補クローズがなくなるまで，以上の過程をくりかえし，なくな

れば，O1は失敗する。高さ1のor節は，それに下っているすべてのユニット節のみが，その解の対象となり，実際にユニファイが成功したもののみが解となるので，上の計算過程は，すべての可能な解を見つけ出している。ゆえに，この場合も定理が成り立つ．

(3). 深さが2NのANDプロセス（インダクション・ステップ）

帰納法の仮定は，つぎの通りである．

「高さが2N-1あるいはそれ以下の任意のORプロセスは，すべての可能な解を，それが起動される毎に1つづつ生成し，最後は"失敗"で終る」

このインダクション・ステップの証明には，つぎのレンマが必要である．

レンマ　AOを，高さが2N以下のAND節とし，$G_1, \ldots, G_n$をそのゴール列とする．いま，制御がMにあり，curr-goalが$G_i$であるとする。さらに，その時点でのB-stackの状態を$\alpha$とし，変数環境を$\beta$とする。するとAMの実行を続けることにより，変数環境$\beta$の下で，$G_i \wedge \cdots \wedge G_n$のすべての可能な解を，AMが再起動される毎に1つづつ生

成し，$G_i$が失敗した直後にⓁに来た時点でのB-stackの状態はαになっている。

インダクション・ステップの証明は，このレンマのiを1とすればよい。

レンマの証明は，つぎの通りである。まず，つぎの補題が成り立つ。

補題 Mにおいて B-スタックのトップにあるORプロセスがゴール $G_i$ を担当しているとする。そのとき，curr_goal は $G_{i+1}$ である。

この補題は，"while fail" ループについての数学的帰納法により，容易に証明できる。レンマの証明も数学的帰納法によるが，$G_1, \cdots, G_i, \cdots, G_n$ のiについての逆順で行う。

a) ベース。($i=n$)

curr_goal は $G_n$ である。Aφを実行すると，$G_n$ を達成するためのORプロセス $ORP_n$ が生成され，起動される。$ORP_n$ が成功すると $ORP_n$ はB-stackへつまれ，Aφは$E_2$で中断する。つぎに再起動されると，Ⓛに行き，B-stack

から ORPn を pop up し，それを再起動する。ORPn から解が生成される間，この同じ過程をくりかえす。ところが，もとの定理の証明中の帰納法の仮定により，ORPn はすべての可能な解を生成する（ORPn の深さは 2N−1 以下だから）。ゆえに，プログラムは，環境 β の下で，すべての可能な解を生成する。もし ORPn が失敗すると，それは削除され，制御は Ⓛ に行くが，そのときの B-stack の状態は，ORPn を pop up した状態であるが，それは，もともと ORPn を push する前の状態 α と同じである。ゆえに，この場合，レンマが成り立つ．

b) インダクション・ステップ

ℓ ≧ j+1 の場合，レンマが成り立つと仮定して，ℓ = j のときにも成り立つことを示す。いま，$G_1 \wedge \cdots \wedge G_{j-1}$ が達成され，制御は，ゴール $G_j$ の実行直前で M にあるものと考える．実行をつづけると，$G_j$ のための OR プロセス $ORP_j$ が生成され，起動される。もしそれが成功すると，それは B-stack に push され，curr_goal が 1 つ進む。そこでのプロセス AM の状態は，帰納法の仮定における状態に等しい。すなわち，$G_1 \wedge \cdots \wedge G_j$ が達成され，制御は，ゴール $G_{j+1}$ の実行直前で M にある。ゆえに，この時点での変数環境の下

で，Aψの以下の実行は，$G_{j+1} \wedge \cdots \wedge G_m$ のすべての可能な解を，それが再起動される毎に，1つづつ生成する。もし $G_{j+1}$ が失敗すると，制御は(L)に行き，B-stackの状態は，$G_{j+1}$ をつむ前の状態になっている。ゆえに，プログラムは，つぎに $ORP_j$ を pop up して，それを再起動する。$ORP_j$ が解を生成する間，Aψは，この過程をくり返す。$ORP_j$ は，すべての可能な解を生成することが（定理1の帰納法の仮定から）知られており，$G_j$ の各解について，Aψは $G_{j+1} \wedge \cdots \wedge G_m$ の可能な解をすべて生成するので，Aψは，$G_j \wedge G_{j+1} \wedge \cdots \wedge G_m$ の（初期環境 $\beta$ の下での）すべての可能な解を生成する。最後に $ORP_j$ が失敗すると，Aψは $ORP_j$ を削除し，つぎに制御が(L)に達した時点でのB-stackの状態は，$ORP_j$ を push する前の状態に戻っている。ゆえに，この場合もレンマが成り立つことが分かる。

(4) 深さが $2N+1$ の OR プロセス（インダクション・ステップ）

本証明は，深さが1の OR プロセスの証明と本質的に変らないので，ここでは省略する。

## 3.2 ユニフィケーション部分

コピーによるユニフィケーションアルゴリズム[Bru80]を下記に示す。この図で、X∅はゴールのパラメータ、Xは選択されたクローズのヘッドのパラメータである。X∅(x)が変数であれば、親(3)のプロセスの中にその変数の値を記憶するスロットXX∅(XX)が必ず存在する。また、あるstructureがコピーされるとき、その中に含まれる変数は、対応するスロットとリンクされ、同一の値を持つことが保証される。

このアルゴリズムで、X∅を現在の環境で評価した値は10と、このプロセス内のスロットによって構成できる。またXについても同様である(証明は略す)

さらに、structureを結ぶポインタは常にプロセスからheapへはcopy stackへ向う。しかも、copy stack中の情報が消えるのはbacktrackの時のみである。従って、プロセスが消滅しても必要なstructureは消えることなく、必ずアクセスすることができる。(structure sharingでは、他のプロセス内の情報を参照するため、determinateに成功したプロセスを消滅させると、必要な情報をアクセスすることができなくなることがある)

## 4. セミ・コルーチン・インタプリタとの等価性

[田村 82] は，Prolog インタプリタがセミ・コルーチンによって記述できることを示した。本稿で与えたコルーチン・インタプリタは，本質的に田村等のセミ・コルーチン・インタプリタと同じである。田村等は，AND プロセスは，実はプロセスである必要がないことを示した。その記述は，さらに AND プロセスをループでなく，再帰呼出しを用いて実現している。田村等の AND プロセスをループ表現にしたものと，我々のコルーチン・インタプリタを比較したのが，Fig. 4 である。

Fig. 4 (a) は，我々のコルーチン・インタプリタでの制御の流れを示している。図の丸で囲んだ部分は，制御を中継しているだけで，これを除くことができる。そうしたのが，Fig. 4 (b) であり，これは田村等のセミ・コルーチン・インタプリタの制御の流れになっている。

我々のコルーチン・インタプリタでは，制御の流れが，隣り合う AND プロセスと OR プロセス間にしかないので，理解が容易となり，検証も，その分，容易になっていると思われる。田村等のセミ・コルーチン・インタプリタは，我々のコルーチン・インタプリタの一種の改良になっていると考えられるであろう。

## 5. おわりに

本論文では，コルーチンに基づくPrologインタプリタの記述と，その正当性の証明を与えた。本インタプリタの記述に用いたPAD表現の上で，カットの導入，決定過程におけるB-stackのpop upによる最適化などが，容易に表現できる（[Furukawa 82]）．さらに，並列インタプリタへの拡張も可能であることも，そこで示した。

最も重要な最適化はtail recursion programを，スタックを使わずに実行するような仕組の導入であるが，それは，すぐ後で削除してもよいプロセスを再利用することで達成できよう。

本研究により，Prologインタプリタの仕組がより容易に理解され，各種の研究がさらに進むことを期待する。

## 参考文献

[Bru 80] Bruynooghe M. "The Memory Management of Prolog Implementation" Logic Programming Workshop

[Furukawa 82] Furukawa K et al. "Prolog Interpleter Based on Concurrent Programming" Prolog Conference

[田村 82] 田村直之 神戸大修士論文

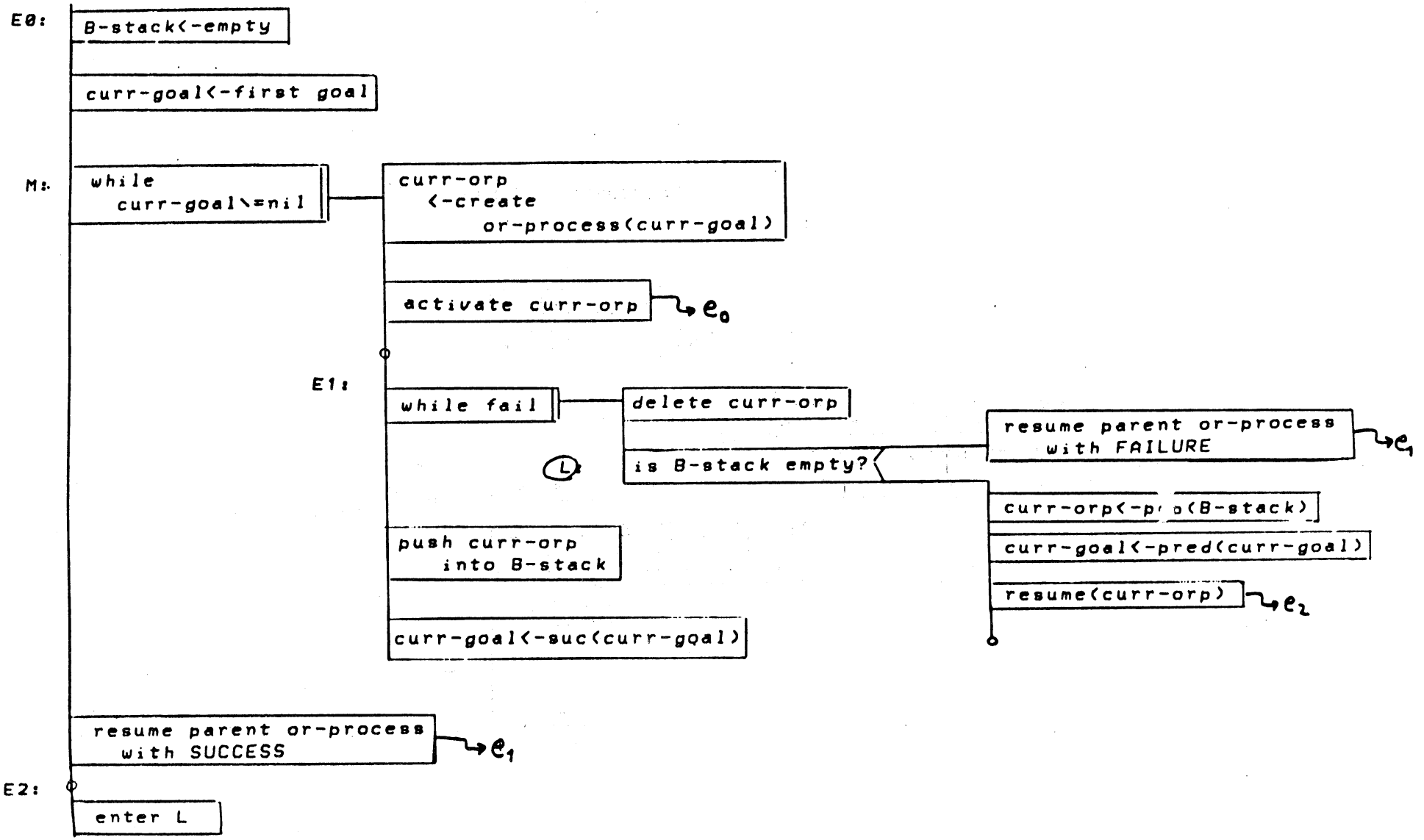


Fig.1. AND-PROCESS(GOALS)

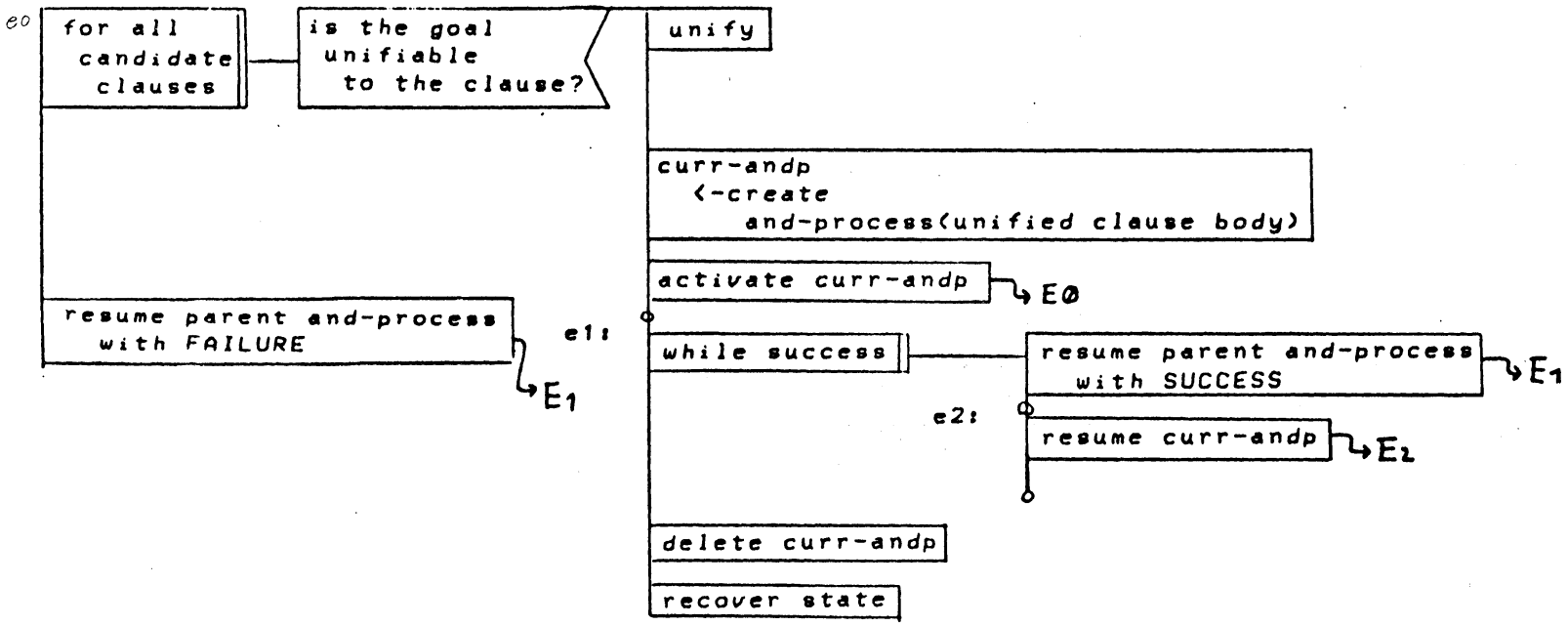


Fig.2. OR-PROCESS(GOAL)

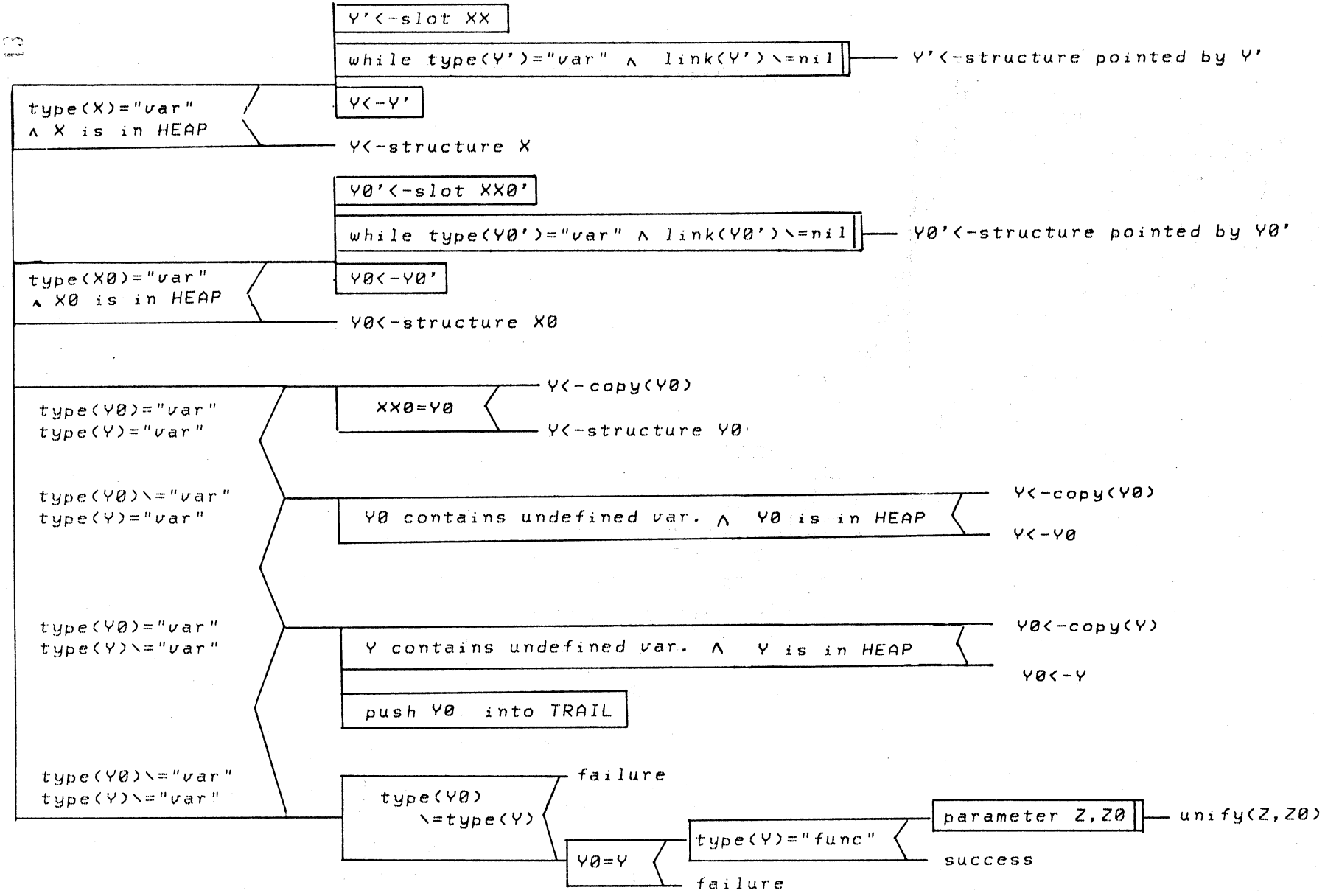


FIG.3 UNIFICATION ALGORITHM USING COPY

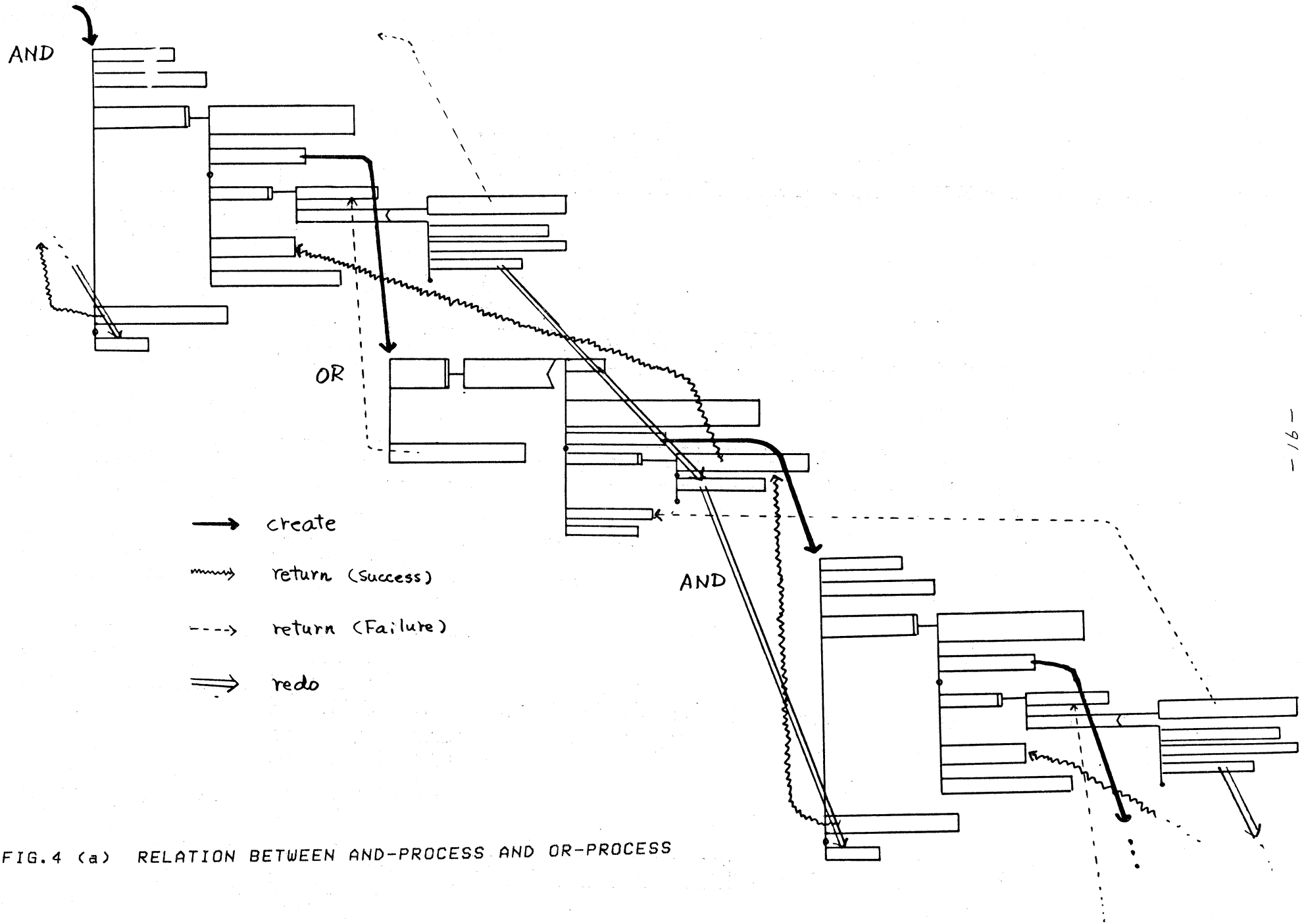


FIG.4 (a) RELATION BETWEEN AND-PROCESS AND OR-PROCESS

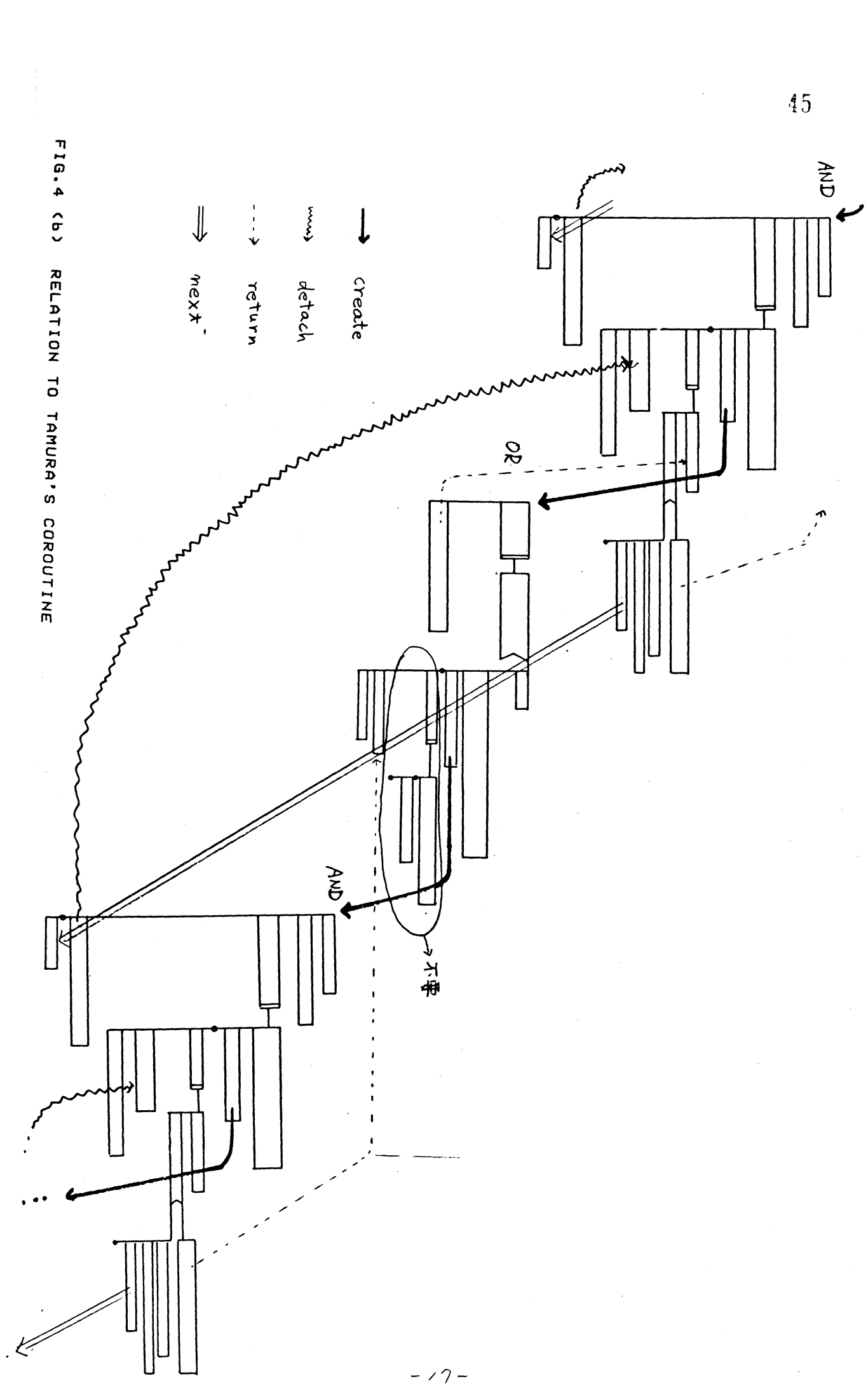


FIG.4 (b) RELATION TO TAMURA'S COROUTINE